

REMARKS/ARGUMENTS

Amendments were made to the specification to update the cross reference to related applications. No new matter has been added by any of the amendments to the specification.

Claims 1-33 are pending in the present application. Claims 1-3, 5-7, 10-24, 27-29, and 32-33 were amended. No claims were added or cancelled. Support for the amendments to the claims may be found in the Specification at least on page 14, lines 1-7, page 25, lines 10-30, page 26, lines 3-4, page 26, lines 8-21; page 27, lines 1-6; page 41, lines 15-26; page 75, lines 24-25; and page 107, lines 1-8. Reconsideration of the claims is respectfully requested.

I. Examiner Interview

Applicant thanks Examiner Truong for all the courtesies extended Applicant's representative during the November 20, 2007 telephone interview. During the interview, Applicant's representative discussed the prior art of record and the manner in which *Sato* in view of *Krishnaiyer* fails to teach or suggest the features recited in the presently claimed invention in claim 1. The Examiner indicated that she would consider the arguments and amendments when submitted. The arguments discussed as well as additional reasons that the claims are not obvious over the cited prior art references are set forth in the remarks below.

II. 35 U.S.C. § 103, Obviousness

The Examiner has rejected claims 1-33 under 35 U.S.C. § 103 as being unpatentable over *Sato et al.* (US 7,020,808) hereinafter referred to as *Sato* in view of *Krishnaiyer et al.* (US 2004/0123041) hereinafter referred to as *Krishnaiyer*. This rejection is respectfully traversed.

The Examiner bears the burden of establishing a *prima facie* case of obviousness based on prior art when rejecting claims under 35 U.S.C. § 103. *In re Fritch*, 972 F.2d 1260, 23 U.S.P.Q.2d 1780 (Fed. Cir. 1992). The prior art reference (or references when combined) must teach or suggest all the claim limitations. *In re Royka*, 490 F.2d 981 180 USPQ 580 (CCPA 1974). In determining obviousness, the scope and content of the prior art are determined; differences between the prior art and the claims at issue are... ascertained; and the level of ordinary skill in the pertinent art resolved. Against this background the obviousness or non-obviousness of the subject matter is determined. *Graham v. John Deere Co.*, 383 U.S. 1 (1966). Often, it will be necessary for a court to look to interrelated teachings of multiple patents; the effects of demands known to the design community or present in the marketplace; and the background knowledge possessed by a person having ordinary skill in the art, all in order to determine whether there was an apparent reason to combine the known elements in the fashion claimed by the patent at issue. *KSR*

Int'l. Co. v. Teleflex, Inc., No. 04-1350 (U.S. Apr. 30, 2007). Rejections on obviousness grounds cannot be sustained by mere conclusory statements; instead, there must be some articulated reasoning with some rational underpinning to support the legal conclusion of obviousness. *Id.* (citing *In re Kahn*, 441 F.3d 977, 988 (CA Fed. 2006)).

II.A. Independent Claim 1

In rejecting claim 1, the Examiner states:

In regard to claim 1, Sato et al. teach a method in a data processing system for gathering performance information associated with executing instructions, the method comprising:
executing instructions in an application by a processor in the data processing system (*events occurred in the processor when the performance of the processor or the computer system is measured or tuned, abstract*);
detecting an indicator associated with an instruction in the executing instructions during execution of the instructions (*when the event condition are satisfied, col. 7 lines 50-54*), wherein the indicator is a data value (*event ID, Fig. 9, 431*) retrieved from memory during execution of the instructions (*each event signal is sent from the processor to the event selector whenever a specific event occurs, col. 7 lines 42-47*), and wherein the indicator specifies counting of events that are associated with execution of the instructions (*execution of a jump instruction or the like, col. 7 lines 43-47*) or counting of events that are associated with accesses to memory locations (*occurrence of memory read, col. 7 lines 43-47*);
sending a signal indicating that the instruction associated with the indicator is being executed (*event signal is sent from processor to event selector whenever a specific event occurs, fig. 1 module 31, col. 7, lines 42-47*);
responsive to receiving the signal, counting, by a functional unit of the processor, events that occur within the data processing system as specified by the indicator to form counted events (*when event condition is satisfied, the event selector sends a signal to the counter controller to instruct the counter controller to increment the count value, fig. 1-3 1-33, col. 7 lines 50-54*),
wherein the indicator enables counting of events on a per instruction (*execution of a jump instruction or the like, col. 7 lines 43-47*) or a per memory location basis (*occurrence of memory read, col. 7 lines 43-47*), and wherein only events specified by indicators are counted (*specific events, col. 7 lines 43-47*); -
retrieving a count value that represents the counted events to form the performance information (*user takes out collected data from the value retaining unit to obtain information, fig. 4, 34, col. 9 lines 35-38*);
Sato et al. does not teach a method in a processing system for gathering performance information comprising of selecting, by the application, an execution path within a set of instructions based on the count value in the performance information, wherein a branch instruction that selects the execution path is contained in an object code block that contains the instructions that were executed while detecting the indicator, and wherein the step of selecting an execution path further comprises: executing instructions that determine if the count value satisfies a first condition; and branching to execute a first set of instructions in response to a determination that the count value satisfies the first condition.

Krishnaiyer et al. teach the method of adaptive prefetch for irregular access pattern by implementing irregular load determination module to check for a selected loop to determine if there is one irregularly accessed load (paragraph 0022) and if condition $l=true$ then prefetch else indicate no prefetch for the loop and execute next instruction (paragraph 0024-0028).

It would have been obvious to modify the method of Sato et al. by adding Krishnaiyer et al. adaptive prefetching for irregular access patterns. A person of ordinary skill in the art at the time of applicant's invention would have been motivated to make the modification because it would improve performance of software applications by reducing memory latency (paragraph 0004).

Office Action, dated September 4, 2007, pages 2-4.

The proposed combination does not teach or suggest all of the features of amended claim 1.

Claim 1, as amended, is as follows:

A method in a data processing system for gathering performance information associated with executing instructions, the method comprising:

executing instructions in an application by a processor in the data processing system;

detecting an indicator associated with an instruction in the executing instructions during execution of the instructions, wherein the indicator is a data retrieved from memory during execution of the instructions, and wherein the indicator specifies counting of events that are associated with execution of the instructions or counting of events that are associated with accesses to memory locations;

sending a signal indicating that the instruction associated with the indicator is being executed to a functional unit in the processor, wherein the functional unit in the processor counts events for instructions in the executing instructions associated with the indicators, and wherein events associated with instructions without indicators are not counted;

responsive to receiving the signal, counting, by the functional unit of the processor, events that occur within the data processing system as specified by the indicator to form counted events, wherein the indicator enables counting of events on a per instruction or a per memory location basis, and wherein only events specified by indicators are counted;

retrieving a count value that represents the counted events to form the performance information; and

selecting, by the application, an execution path within a set of instructions based on the count value in the performance information gathered by the functional unit in the processor in accordance with the indicators associated with the executing instructions, wherein a branch instruction that selects the execution path is contained in an object code block that contains the instructions that were executed while detecting the indicator, and wherein the step of selecting an execution path further comprises:

executing instructions that determine if the count value satisfies a first condition;

branching to execute a first set of instructions in response to a determination that the count value satisfies the first condition; and

branching to execute a second set of instructions in response to a determination that the count value satisfies a second condition.

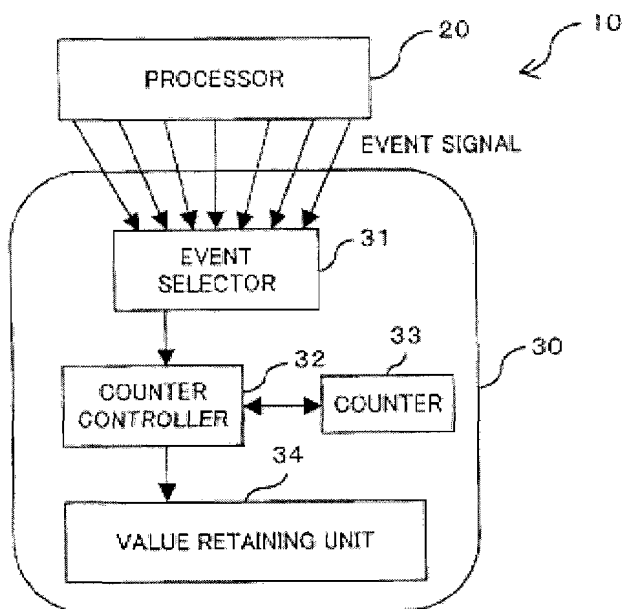
Independent claim 12 and 23 recites similar subject matter. All claim limitations must be considered, especially when missing from the prior art. The proposed combination of *Sato* and

Krishnaiyer, considered as a whole, does not teach or suggest the claimed features of sending a signal to a functional unit of the processor or selecting an execution path.

II.A.i Sending a signal indicating that the instruction associated with the indicator is being executed to a functional unit in the processor, wherein the functional unit in the processor counts events for instructions in the executing instructions associated with the indicators, and wherein events associated with instructions without indicators are not counted.

The combination of *Sato* and *Krishnaiyer*, considered as a whole, does not teach or suggest the claimed features of sending a signal indicating that the instruction associated with the indicator is being executed to a functional unit in the processor, wherein the functional unit in the processor counts events for instructions in the executing instructions associated with the indicators, and wherein events associated with instructions without indicators are not counted, as in amended claim 1.

The Examiner cites to figure 1 of *Sato* which illustrates:



As shown in Figure 1 of *Sato*, a processor sends event signals to an event selector. The portion of *Sato* describing Figure 1 states:

The event selector 31 receives an event signal from each part of the processor 20 to function as follows: Each event signal is sent from the processor 20 to the event selector 31 whenever a specific event (for example, elapse of one clock, occurrence of memory read, execution of a jump instruction, or the like) occurs. Event conditions are set to the event selector 31 from the outside. Various event conditions can be set, as occurrence of an event A, simultaneous occurrence of events A and B, etc. When the event condition is satisfied, the event selector 31 sends a signal to the counter controller 32 to instruct the counter controller 32 to increment the count value of the counter 33 by one.

Sato, column 7, lines 42-53.

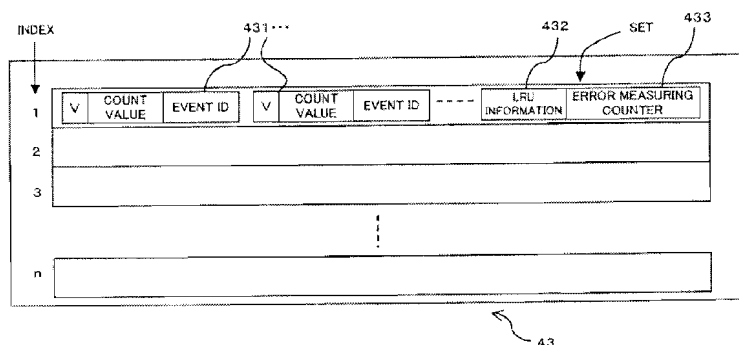
The event selector receives the event signal from the processor when a specific event occurs. The event signal is not sent **to the processor**. In contradistinction, the invention in claim 1 “**sending a signal** indicating that the instruction associated with the indicator is being executed **to a functional unit in the processor**.” In other words, *Sato* teaches an event selector that receives event signals from a processor. *Sato* does not send a signal to a functional unit in the processor or receive a signal by a functional unit in the processor. In addition, *Sato* sends the signal whenever a specific event occurs, such as elapse of one clock, occurrence of a memory read, or execution of a jump instruction. In contrast, the signal in claim 1 is sent to the functional unit in the processor to indicate that an instruction associated with an indicator is executed. *Sato* does not teach, suggest, or even mention a signal indicating that an instruction associated with an indicator is being executed.

As shown above, *Sato* sends the event signal from the processor when **specific events** occur, without regard for whether the event is associated with an instruction having an indicator or whether the event is associated with an instruction without an indicator. The signal in claim 1 indicates instructions associated with an indicator. The signal in claim 1 is only sent when an instruction associated with an indicator is executed.

Finally, *Sato* sends a signal to the counter controller to instruct the counter controller to increment the count value of the counter by one when an event condition is satisfied. *Sato* states that event conditions may be occurrence of an event A or simultaneous occurrence of events A and B. An event is not an indicator associated with an instruction. *Sato* counts events when an event condition occurs. The event condition described by *Sato* does not teach or suggest an indicator that is a data. Thus, *Sato* counts events without regard to indicators associated with instructions. *Sato* does not teach that specific events are not counted if the specific events are associated with instructions without indicators.

The Examiner alleges that *Sato* teaches an indicator in the Event ID shown in Figure 9, item 431 which illustrates as follows:

FIG. 9



Here, *Sato* illustrates a value retaining unit. The portion of *Sato* explaining item 431 in Figure 9 states:

The count value retaining area (a retaining area) 431 has a predetermined number of records being able to hold count values therein. Each record holds a set of an event ID, and the number of events having occurred (a count value) corresponding to the event ID, along with a valid/invalid bit (mentioned as V in the drawing) representing whether the record is valid or invalid.

Sato, column 16, lines 56-62.

As shown above, the event ID of *Sato* is an identifier for an event. The count value retaining area holds a count value for each event ID. However, as discussed above, an event is not an indicator associated with an instruction as claimed in claim 1. Claim 1 states “the functional unit in the processor counts events for instructions in the executing instructions associated with the indicators.” It would make no sense to interpret the term “indicator” so broadly as to encompass events when claim 1 clearly states that the functional unit **counts events for instructions associated with the indicators**. Moreover, the event ID of *Sato* is an identifier that corresponds to an event count in a value retaining unit. In contrast, the indicator of claim 1 is associated with an instruction executing in the processor. A separate functional unit counts events for instructions associated with the indicator. Therefore, the event ID of figure 9 cannot teach or suggest the indicator of claim 1 in the present invention.

Therefore, *Sato* fails to teach or suggest “sending a signal indicating that the instruction associated with the indicator is being executed to a functional unit in the processor, wherein the functional unit in the processor counts events for instructions in the executing instructions associated with the indicators, and wherein events associated with instructions without indicators are not counted” as is claimed in amended claim 1.

II.A.ii Selecting, by the application, an execution path within a set of instructions based on the count value in the performance information gathered by the functional unit in the processor in accordance with the indicators associated with the executing instructions, wherein a branch instruction that selects the execution path is contained in an object code block that contains the instructions that were executed while detecting the indicator, and wherein the step of selecting an execution path further comprises: executing instructions that determine if the count value satisfies a first condition; branching to execute a first set of instructions in response to a determination that the count value satisfies the first condition; and branching to execute a second set of instructions in response to a determination that the count value satisfies a second condition.

The Examiner failed to state a prima facie obviousness rejection of claim 1, because the proposed combination of references considered as a whole, does not teach or suggest the feature of selecting, by

the application, an execution path within a set of instructions based on the count value in the performance information gathered by the functional unit in the processor in accordance with the indicators associated with the executing instructions, wherein a branch instruction that selects the execution path is contained in an object code block that contains the instructions that were executed while detecting the indicator, and wherein the step of selecting an execution path further comprises: executing instructions that determine if the count value satisfies a first condition; branching to execute a first set of instructions in response to a determination that the count value satisfies the first condition; and branching to execute a second set of instructions in response to a determination that the count value satisfies a second condition, as recited in claim 1.

The Examiner acknowledges that *Sato* fails to teach or suggest this feature. However, the Examiner believes that *Krishnaiyer* teaches this feature where *Krishnaiyer* teaches adaptive prefetch for irregular access patterns by implementing irregular load determination module to check for a selected loop to determine if there is one irregularly accessed load at paragraph 0022 which states:

In embodiments of the present invention, an irregular load determination module in the computer program product may check a selected loop to determine if there is one irregularly accessed load inside the selected loop. An irregularly accessed load may be defined as a load where the computer program product, e.g., the compiler, may not be able to determine the address of the memory load for future iterations of the loop statically (during compilation from source code to output code). If the selected loop does not include the one irregularly accessed load inside the selected loop, the computer program product may not insert the conditional adaptive prefetch code into the output code for the selected loop.

Here, *Krishnaiyer* describes a load where the compiler may not be able to determine the address of the memory load for future iterations of the loop statically. Adaptive prefetch code is not inserted into the output code for the selected loop if the loop doesn't contain the one irregularly accessed load. *Krishnaiyer* merely teaches adaptive prefetch code. Although *Krishnaiyer* may teach determining whether a loop has an irregularly accessed load and inserting or not inserting conditional adaptive prefetch code, *Krishnaiyer* does not select an execution path based on a count value in performance information gathered in accordance with indicators. In fact, *Krishnaiyer* does not teach, suggest, or even mention indicators associated with executing code. Moreover, as discussed above, *Sato* fails to teach the indicators of claim 1.

In addition, the cited portion of the reference does not insert conditional adaptive prefetch code into output code for a loop based on whether there is an irregularly accessed load inside the loop. Moreover, *Krishnaiyer* does not select an execution path. Rather, this portion of *Krishnaiyer* involves inserting prefetch code rather than selecting an execution path.

The Examiner also cites to paragraphs [0024]-[0028] which states:

```
[0024] If (condition 1=true) (** condition 1 determines if there is a data prefetch **)
then
[0025] {prefetch (first field of data for future iteration of loop) prefetch (ninth field of
data for future iteration of loop)}
[0026] else
[0027] {indicate no prefetch for the loop}
[0028] Execute next instruction
```

In this section of *Krishnaiyer*, conditional prefetch code including a loop is shown. The loop performs a data prefetch if condition 1 is true. The loop performs no prefetch for the loop if condition 1 is not true. Although *Krishnaiyer* discloses a loop function with a conditional if-then-else statement, such code statements do not teach or suggest selecting an execution path within a set of instructions **based on a count value** in the performance information gathered by the functional unit **in accordance with the indicators** where the functional unit in the processor counts events for instructions in the executing instructions associated with the indicators, and where events associated with instructions without indicators are not counted. Thus, the proposed combination of *Sato* and *Krishnaiyer* fails to teach or suggest “[s]electing, by the application, an execution path within a set of instructions based on the count value in the performance information gathered by the functional unit in the processor in accordance with the indicators associated with the executing instructions, wherein a branch instruction that selects the execution path is contained in an object code block that contains the instructions that were executed while detecting the indicator, and wherein the step of selecting an execution path further comprises: executing instructions that determine if the count value satisfies a first condition; branching to execute a first set of instructions in response to a determination that the count value satisfies the first condition; and branching to execute a second set of instructions in response to a determination that the count value satisfies a second condition.”

II.B. Independent Claim 5

With regard to claim 5, the Examiner states:

In regard to claim 5, *Sato et al.* teach a method in a data processing system for gathering performance information associated with executing instructions, the method comprising:
compiling source code statements to generate instructions for a processor in the data processing system (*set an event that is an object of measurement, fig. 10, S51*);
generating indicators to be processed during execution of the instructions (*processor performs information processing which is an object to measurement of the number of events having occurred, col. 7 lines 25-28*), wherein the indicators are data values (*event ID, fig. 9, 431*) that are retrieved from memory

during execution of the instructions that specify counting of events that are associated with execution of the instructions or are data values that are retrieved from memory during execution of the instructions that specify counting of events that are associated with accesses to memory locations (*when event condition is satisfied, the event selector sends a signal to the counter controller to instruct the counter controller to increment the count value, fig. 1-31 -33, col. 7 lines 50-54*), and wherein the indicators specify counting of events on a per instruction (*execution of a jump instruction or the like, col. 7 lines 43-47*) or a per memory location basis (*occurrence of memory read, col. 7 lines 43-47*) such that only the events specified by the indicators are counted (*specific events, col. 7 lines 43-47*) to form the performance information (*performance measuring counter, fig. 1, 30*); Sato et al. does not teach a method in a processing system for gathering performance information comprising of generating instructions to select an execution path within a set of instructions based on a count value in the performance information that represents counted events, wherein a branch instruction that selects the execution path is contained in an object code block that contains the instructions that are executed while the indicators are processed. Krishnaiyer et al. teach the method of adaptive prefetch for irregular access pattern by implementing irregular load determination module to check for a selected loop to determine if there is one irregularly accessed load (*paragraph 0022*) and if condition l=true then prefetch else indicate no prefetch for the loop and execute next instruction (*paragraph 0024-0028*).

Refer to claim 1 for motivational statement.

Office Action dated September 4, 2007, pages 6-7.

The proposed combination does not teach or suggest all of the features of amended claim 5.

Claim 5, as amended, is as follows:

A method in a data processing system for gathering performance information associated with executing instructions, the method comprising:
 compiling source code statements to generate instructions for a processor in the data processing system;
 generating, by the compiler, indicators to be processed by the processor during execution of the instructions, wherein the indicators are data values that are retrieved from memory during execution of the instructions that specify counting of events that are associated with execution of the instructions or are data values that are retrieved from memory during execution of the instructions that specify counting of events that are associated with accesses to memory locations, and wherein the indicators specify counting of events on a per instruction or a per memory location basis such that only the events specified by the indicators are counted to form the performance information; and
 generating instructions to select an execution path within a set of instructions based on a count value in the performance information that represents counted events, wherein a branch instruction that selects the execution path is contained in an object code block that contains the instructions that are executed while the indicators are processed.

Independent claims 16 and 27 recite similar subject matter. The proposed combination of *Sato* and *Krishnaiyer*, considered as a whole, does not teach or suggest the claimed features of an indicator, and the steps of generating indicators and generating instructions to select an execution path.

II.B.i. Generating, by the compiler, indicators to be processed by the processor during execution of the instructions, wherein the indicators are data values that are retrieved from memory during execution of the instructions that specify counting of events that are associated with execution of the instructions or are data values that are retrieved from memory during execution of the instructions that specify counting of events that are associated with accesses to memory locations, and wherein the indicators specify counting of events on a per instruction or a per memory location basis such that only the events specified by the indicators are counted to form the performance information.

The Examiner failed to state a prima facie obviousness rejection of claim 5, because the proposed combination of references considered as a whole, does not teach or suggest the feature of a generating, by the compiler, indicators to be processed by the processor during execution of the instructions, wherein the indicators are data values that are retrieved from memory during execution of the instructions that specify counting of events that are associated with execution of the instructions or are data values that are retrieved from memory during execution of the instructions that specify counting of events that are associated with accesses to memory locations, and wherein the indicators specify counting of events on a per instruction or a per memory location basis such that only the events specified by the indicators are counted to form the performance information,” as is claimed in amended claim 5.

The Examiner cites to *Sato* at column 7, lines 25-28 which states:

The processor 20 performs information processing, which is an object of performance measurement (an object of measurement of the number events having occurred).

Sato states the processor performs information processing. *Sato* does not teach or suggest a compiler generates indicators that specify counting of events. The Examiner also cites to Figure 9, item 431, which is shown above. As discussed above, the event ID of *Sato* is merely an identifier corresponding to counted events in a retaining unit. The event ID does not teach or suggest an identifier as is claimed in claim 1. Moreover, even if the event ID of *Sato* could somehow suggest an identifier associated with executing instructions, *Sato* does not teach or suggest that the indicator is generated by a compiler. As discussed above, *Krishnaiyer* also fails to teach or suggest an indicator or an indicator generated by a compiler. Therefore, the combination of *Sato* and *Krishnaiyer* fails to teach or suggest “generating, by the compiler, indicators to be processed by the processor during execution of the instructions, wherein the indicators are data values that are retrieved from memory during execution of the instructions that specify counting of events that are associated with execution of the instructions or are data values that are retrieved from memory during execution of the instructions that specify counting of events that are associated with accesses to memory locations, and wherein the indicators specify counting of events on a per instruction or a per memory location basis such that only the events specified by the

indicators are counted to form the performance information,” as is claimed in amended claim 5. Thus, *Sato* and *Krishnaiyer* fails to teach or suggest each and every feature of claim 5.

II.C. *Sato* teaches away from the presently claimed invention

Sato teaches away from the invention in claim 1. As shown above, *Sato* teaches a processor sending signals to an event selector whenever an event occurs. In contrast, claim 1 requires sending a signal to a function unit **in the processor**. In other words, the functional unit in the processor **receives** the signal and **does not send the signal** outside the processor. In addition, *Sato* counts events once event conditions are satisfied. *Sato* teaches that event conditions are the occurrences of events. *Sato* does not teach counting events or not counting events based on indicators associated with the executing instructions. In contradistinction, the invention in claim 1 counts events for instructions in the executing instructions associated with the indicators. In addition, events associated with instructions without indicators are not counted, as is claimed in claim 1. Thus, one of ordinary skill would not be motivated to modify *Sato* to reach the presently claimed invention because *Sato* teaches away from the presently claimed invention. Thus, the Examiner has failed to state a *prima facie* obviousness with regard to the presently claimed invention in claim 1.

II.D. The Examiner fails to state a sufficient reason to modify the reference

The Examiner bears the burden of establishing a *prima facie* case of obviousness based on prior art when rejecting claims under 35 U.S.C. § 103. *In re Fritch*, 972 F.2d 1260, 23 U.S.P.Q.2d 1780 (Fed. Cir. 1992). The scope and content of the prior art are... determined; differences between the prior art and the claims at issue are... ascertained; and the level of ordinary skill in the pertinent art resolved. Against this background the obviousness or non-obviousness of the subject matter is determined. *Graham v. John Deere Co.*, 383 U.S. 1 (1966). Often, it will be necessary for a court to look to interrelated teachings of multiple patents; the effects of demands known to the design community or present in the marketplace; and the background knowledge possessed by a person having ordinary skill in the art, all in order to determine whether there was an apparent reason to combine the known elements in the fashion claimed by the patent at issue. *KSR Int’l. Co. v. Teleflex, Inc.*, No. 04-1350 (U.S. Apr. 30, 2007). Rejections on obviousness grounds cannot be sustained by mere conclusory statements; instead, there must be some articulated reasoning with some rational underpinning to support the legal conclusion of obviousness. *Id.* (citing *In re Kahn*, 441 F.3d 977, 988 (CA Fed. 2006)).

In the case at hand, no *prima facie* obviousness rejection can be stated because the Examiner failed to state a sufficient reason to modify *Sato* and *Krishnaiyer* in light of the great differences between the cited art and amended claim 1. Specifically, as shown above, *Sato* and *Krishnaiyer* fails to teach or

suggest the feature of an indicator associated with executing code, a compiler generating the indicator, and a functional unit counting events associated with instructions associated with indicators but not counting events for instructions without indicators.

The Examiner failed to state a sufficient reason to modify *Sato* and *Krishnaiyer* because the Examiner's proposed reason for modifying the cited art provides no rational underpinning to support a legal conclusion of obviousness. Regarding a reason to modify *Sato* and *Krishnaiyer* the Examiner states that it would have been obvious to modify the method of *Sato* by adding the adaptive prefetching for irregular access patterns of *Krishnaiyer* because "it would improve performance of software applications by reducing memory latency." However, the Examiner fails to provide a sufficient reason to modify *Sato* and *Krishnaiyer* because the Examiner merely offers a possible advantage for the modification without providing any reason for the modification. In particular, the Examiner fails to provide a reason or rationale underpinning for modifying *Sato* to include an indicator associated with executing code, generate an indicator by a compiler, only counting events for instructions associated with an indicator, or selecting an execution path based on a count value generated in accordance with the indicators. Thus, the Examiner's reason for modifying *Sato* and *Krishnaiyer* provides an insufficient basis for modifying the teachings of the cited art in the manner necessary to reach each and every feature of amended claim 1, especially in the light of the large differences that exist between *Sato* in view of *Krishnaiyer* and amended claims 1 and 5.

For these reasons, the rejection of obviousness vis-à-vis amended claim 5 has been overcome.

III. Independent Claims

Independent claims 12, 16, 23, and 27 have features similar to those presented in claim 1 and claim 5. Therefore, claims 12, 16, 23, and 27 are non-obvious at least for the reasons set forth above.

IV. Dependent Claims

Claims 2-4, 6-11, 13-22, 24-26, and 28-33 depend from claims 1, 5, 12, 16, 23, and 27. Applicants have already demonstrated claims 1, 5, 12, 16, 23, and 27 are not obvious and are therefore in condition for allowance. Therefore, at least by virtue of their dependence on claims 1, 5, 12, 16, 23, and 27 claims 2-4, 6-11, 13-22, 24-26, and 28-33 are not obvious over *Sato* in view of *Krishnaiyer*.

In addition, the dependent claims recite additional combinations of features not taught by the cited art. For example, claim 10 recites:

The method of claim 5 further comprising:
determining, by the compiler, alternative ways to compile a subroutine within the source code statements;

generating a plurality of compiled versions of the subroutine, wherein each compiled version has been compiled differently from other compiled versions of the subroutine; and

placing the plurality of compiled versions of the subroutine in different execution paths to be selected based on the count value.

The Examiner cites to *Krishnaiyer* at paragraph [0011] which states:

In an embodiment of the invention, the computer program product, e.g., the compiler, may determine during compilation of the source code 1) if the loop is accessed a sufficient number of times to achieve a performance gain from active prefetching; and 2) if an irregular load exists in the loop. In this embodiment of the invention, the computer program product may generate code to determine dynamically, i.e., during execution of the output code, whether the irregular load has a predictable access pattern. The computer program product may also insert a prefetch instruction into the output code, the prefetch instruction only being executed if certain conditions are met. This may be referred to as insertion of conditional adaptive prefetch code including a conditional prefetch instruction. If the computer program product determines statically that the loop was accessed a sufficient number of times and that an irregular load exists in the loop, and the generated code identifies that the irregular load has a predictable access pattern, then the conditional prefetch instruction in the inserted conditional adaptive prefetch code may be executed.

Here, *Krishnaiyer* discloses a compiler determining if a loop is accessed a sufficient number of times to benefit from active prefetching and if an irregular load exists. A compiler determining a number of access times and whether an irregular load exists does not teach or suggest a compiler generating a **plurality of compiled versions** of a single subroutine and placing the plurality of compiled versions of the subroutine in different execution paths to be selected based on the count value that is generated using the indicators. Thus, *Krishnaiyer* fails to teach each and every feature of claims 10, 21, and 32.

Therefore, the rejection of claims 1-33 under 35 U.S.C. § 103 has been overcome.

V. Conclusion

It is respectfully urged that the subject application is patentable over *Sato* in view of *Krishnaiyer* and is now in condition for allowance. The examiner is invited to call the undersigned at the below-listed telephone number if in the opinion of the examiner such a telephone conference would expedite or aid the prosecution and examination of this application.

DATE: December 4, 2007

Respectfully submitted,

/Mari A. Stewart/

Mari A. Stewart
Reg. No. 50,359
Yee & Associates, P.C.
P.O. Box 802333
Dallas, TX 75380
(972) 385-8777
Attorney for Applicants